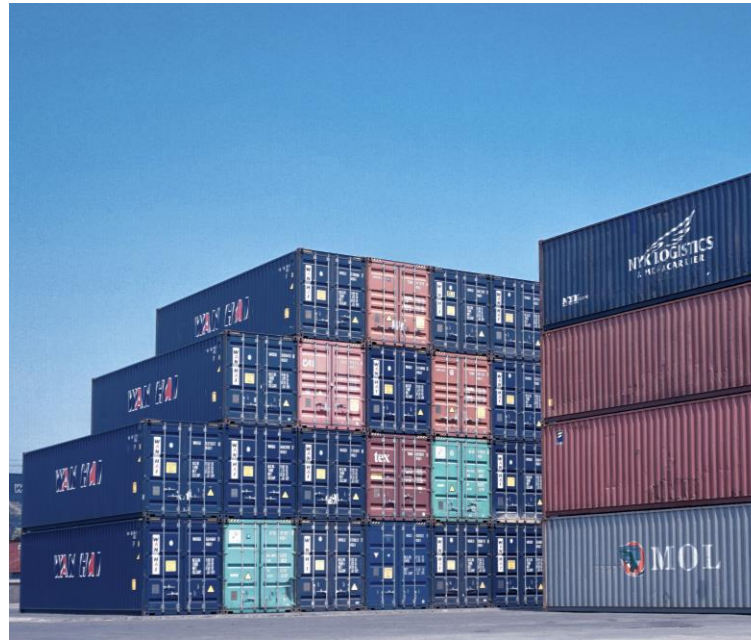


Microservices II: Using Microservices & Discussion

CNAE – Cloud Native Architecture & Engineering



Prof. Stefan Tai
Sebastian Werner, Elias Grünewald, Nicola Leschke, Maria Borges

Information Systems Engineering
TU Berlin

Formal registration deadline: 15.05.2023, 23:59

The registration via moses: <https://moseskonto.tu-berlin.de/moses/modultransfersystem/index.html>

- select "MTS / Module exams registration/deregistration" after logging in
- Select right course and semester!
- instructions with screenshots are only available in [German](#)

Recap: Microservices I

Microservices...

...are **autonomously deployable services** providing a **focused amount** of functionality.

...have **nine characteristics** (according to Fowler).

...are **standardized with respect to interfaces and deployment** procedures, but not things „below the surface“.

...require **organizational structure** and a **DevOps** culture.

- Small teams develop small services and own them as products
- Coordination through service interfaces
 - Changes are communicated, implemented, and added to the interface

Learning Objectives

- Deepening your knowledge about microservice design, with a focus on communication and data management.
- Understand challenges connected to decentralized data management.
- Recapitulate different communication mechanisms and evaluate their suitability for a specific microservice architecture.
- Learn about conflicting software qualities and how to address them in microservice architectures.
- Understand the microservice premium and avoid common pitfalls in microservice design.

Agenda



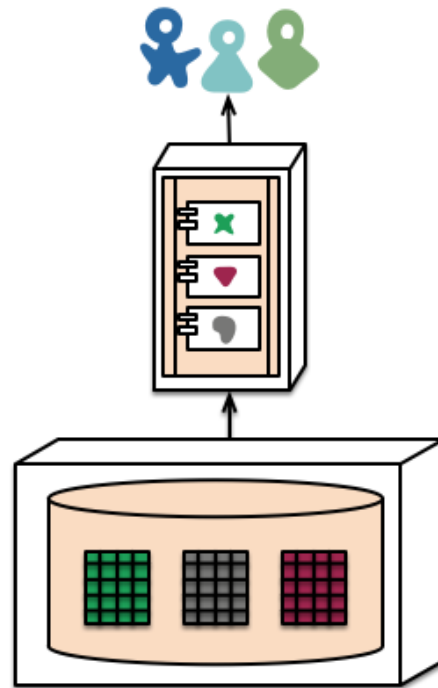
Microservices I: Characteristics and Design Principles

Microservices II: Using Microservices & Discussion

1. Data Management
2. Communication
3. API Design
4. Ifs and Buts of Microservices
5. Orga

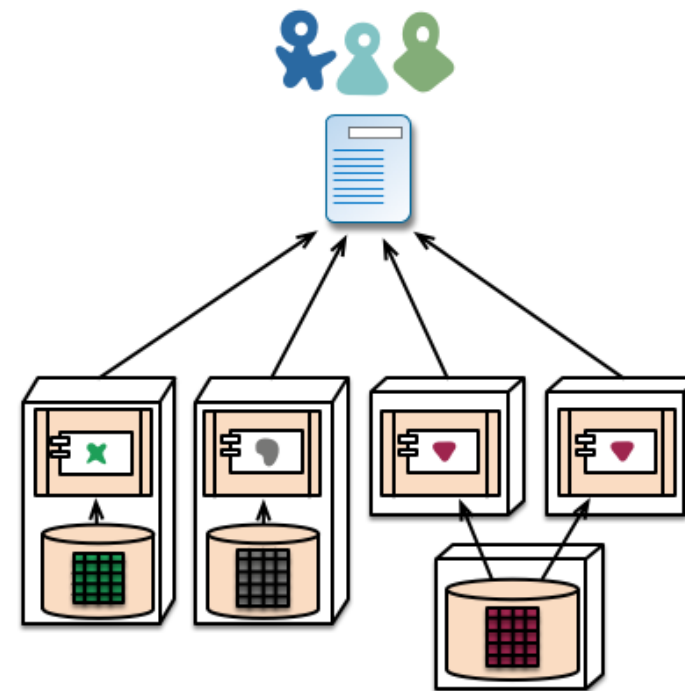
Recall: Microservice Principle #6- Decentralized Data Management

In monoliths, the application uses a shared database.



monolith - single database

Each microservice maintains its own database.



microservices - application databases

Recall: ACID I

Transactions have 4 main properties:

Atomicity → In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are.

Consistency

Isolation

Durability

Recall: ACID II

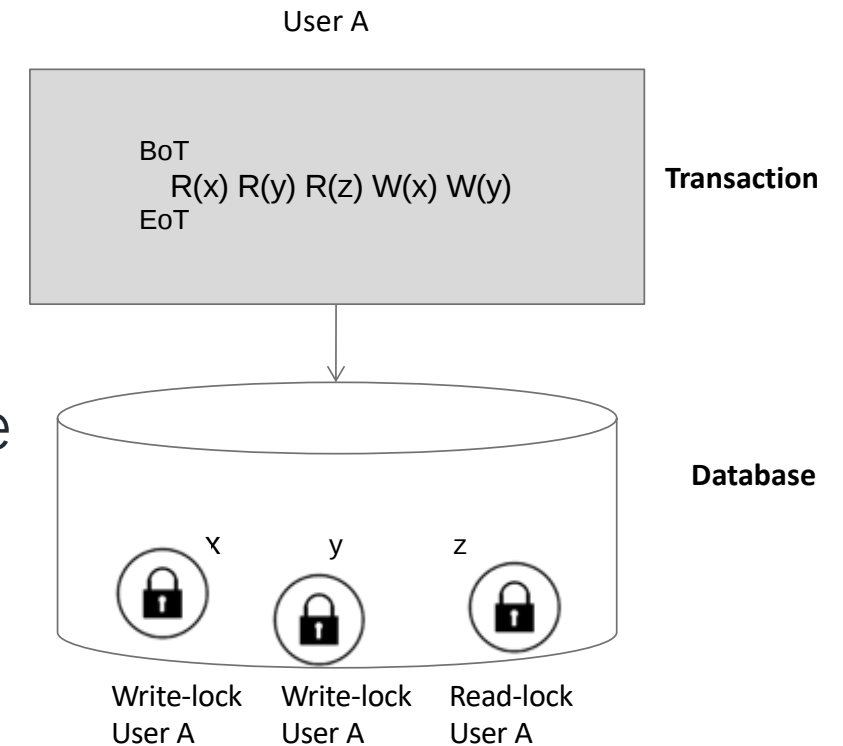
Transactions have 4 main properties:

Atomicity

Consistency

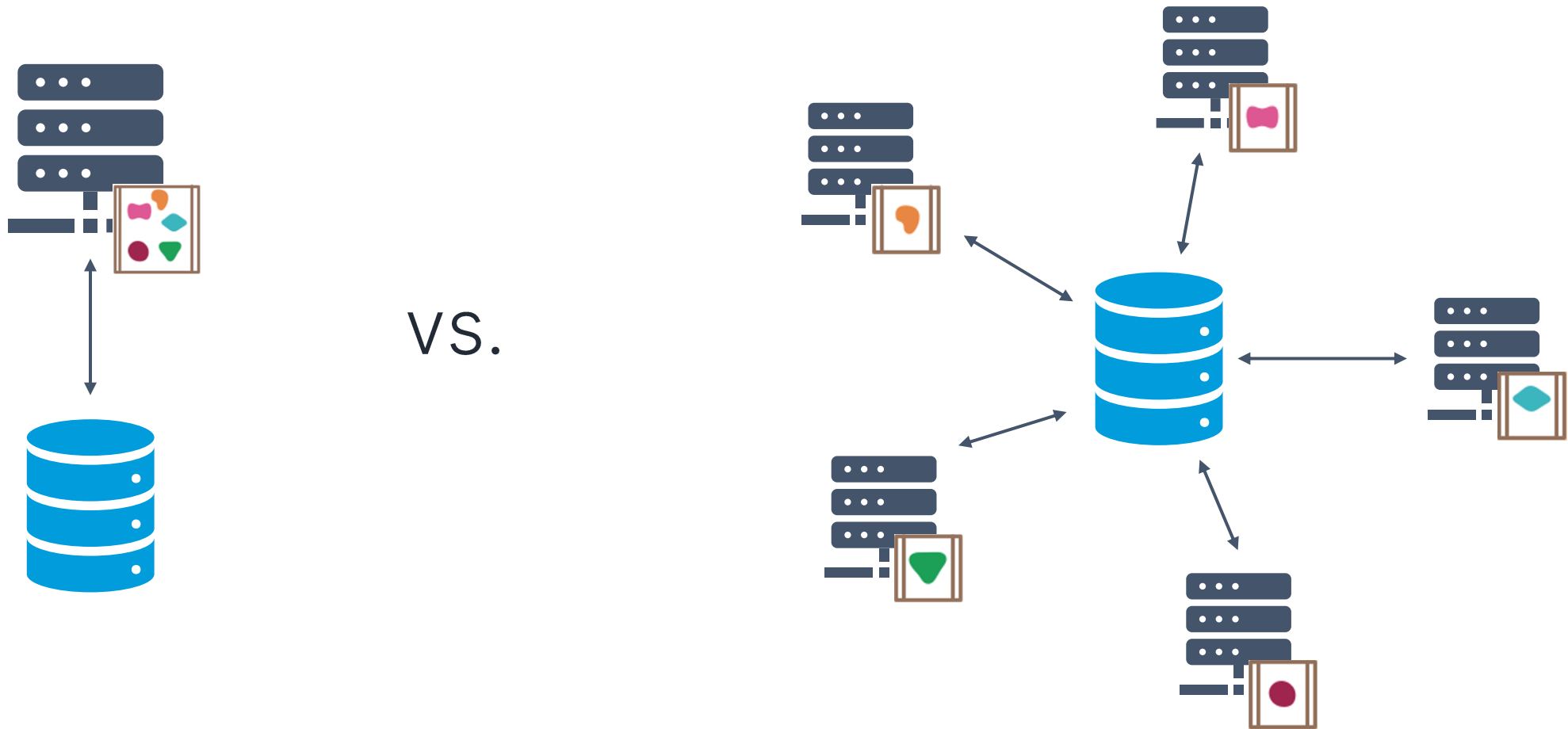
Isolation → When transactions are executed concurrently, the effect should be the same as if they ran independently. Guaranteed by a **concurrency control protocol**.

Durability

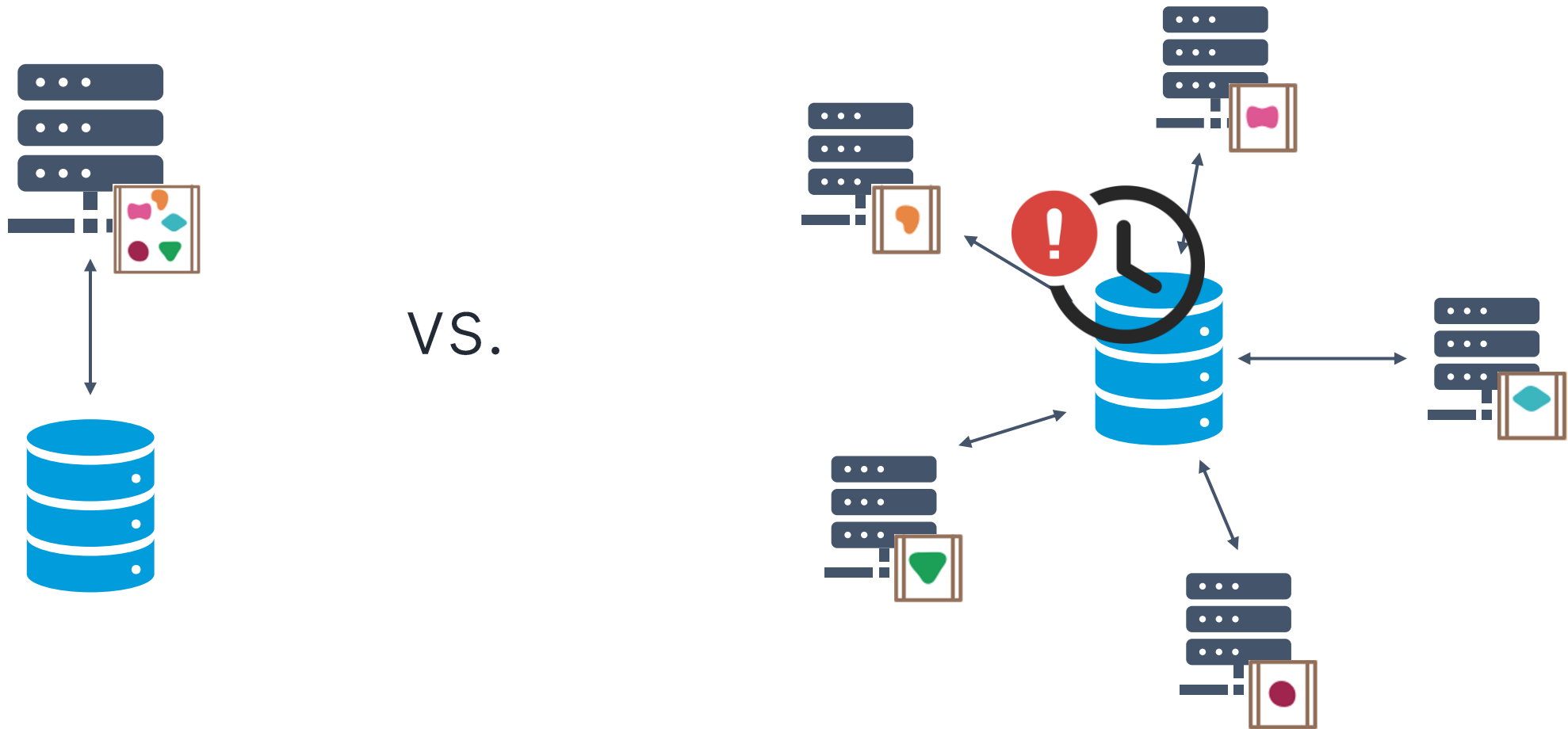


Source: Gustavo Alonso

Transactions + Microservices?



Transactions + Microservices?



Decentralized Data Management



Quick Quiz:

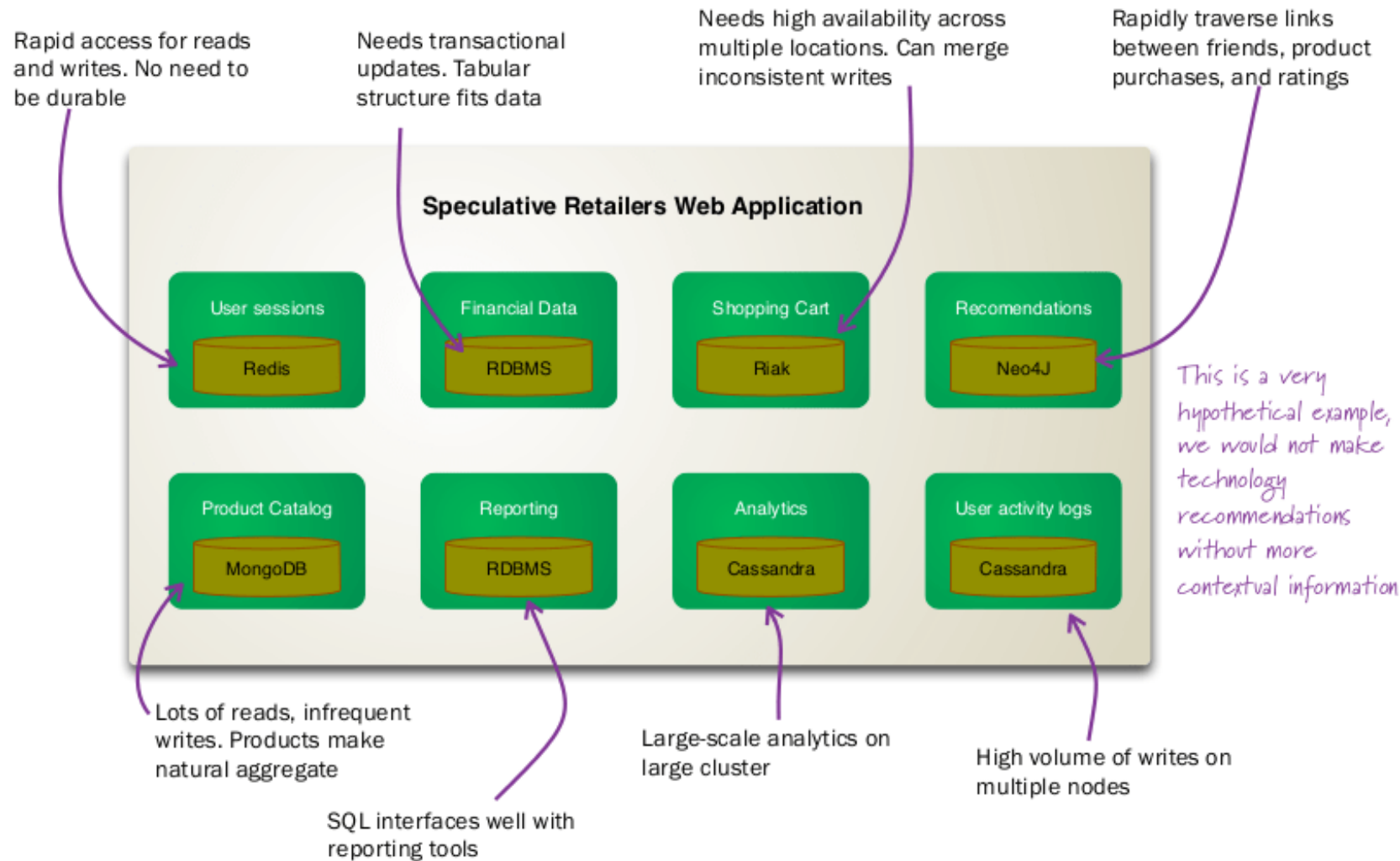
Other advantages of decentralized data management?

Quick Quiz:

Other advantages of decentralized data management?

- Independent Scalability + no single-point-of-failure
- Evolvability - Schema changes don't affect other services
 - Polyglot Persistence

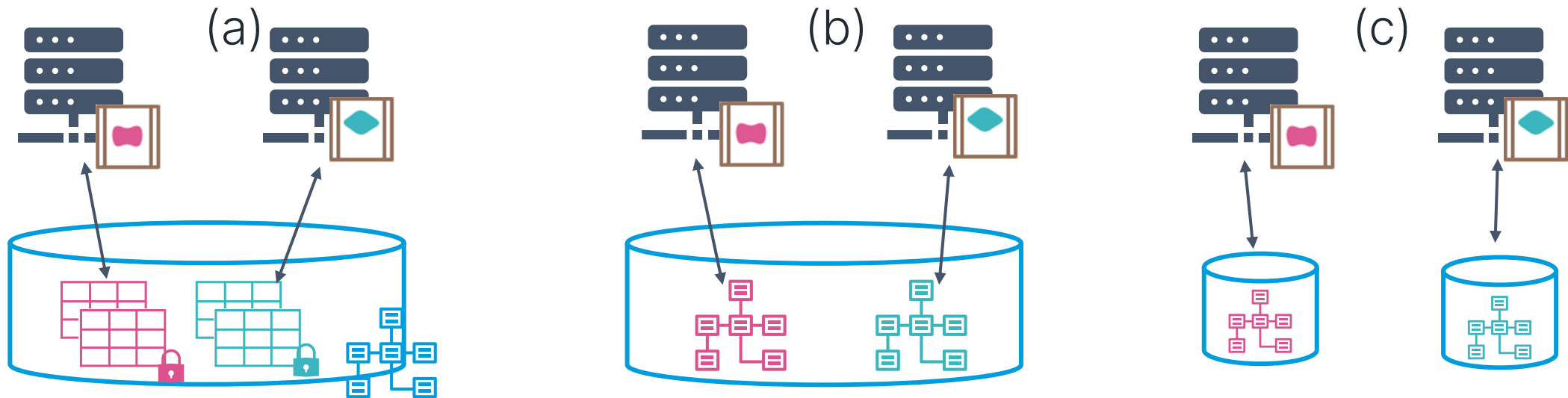
Polyglot Persistence



<https://www.abhishek-tiwari.com/polyglot-persistence-patterns/>

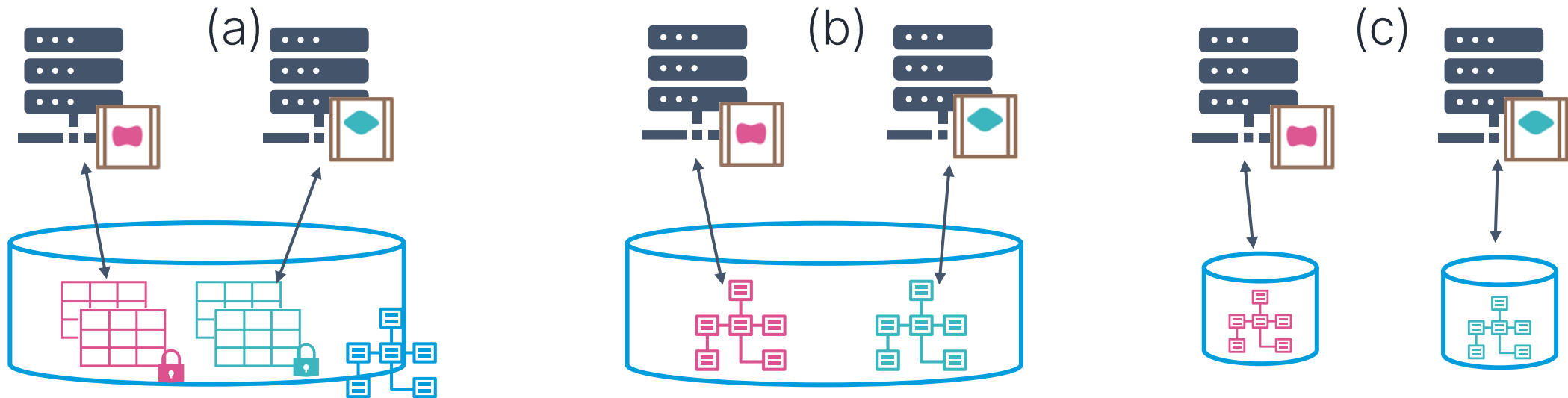
Microservice Data Management - Deployment Patterns

- (a) Private Tables per microservice, sharing a database server and schema
- (b) Schema per microservice, sharing a common database server
- (c) Database server per microservice



Deployment Patterns in Practice – Survey Results

- (a) Private Tables per microservice, sharing a database server and schema → 17%
- (b) Schema per microservice, sharing a common database server → 33%
- (c) Database server per microservice → 43%



Source: doi.org/10.14778/3484224.3484232

What new problems arise due to decentralization?

- Increased Operational Complexity – More databases to set up and maintain
- Cost overhead – Increased infrastructure costs (+ licensing costs)
- Storage overhead due to potential data duplication
- Ensuring data is consistent across different data sources
- Implementing business transactions that span multiple services is not straightforward

Querying Data Across Microservices

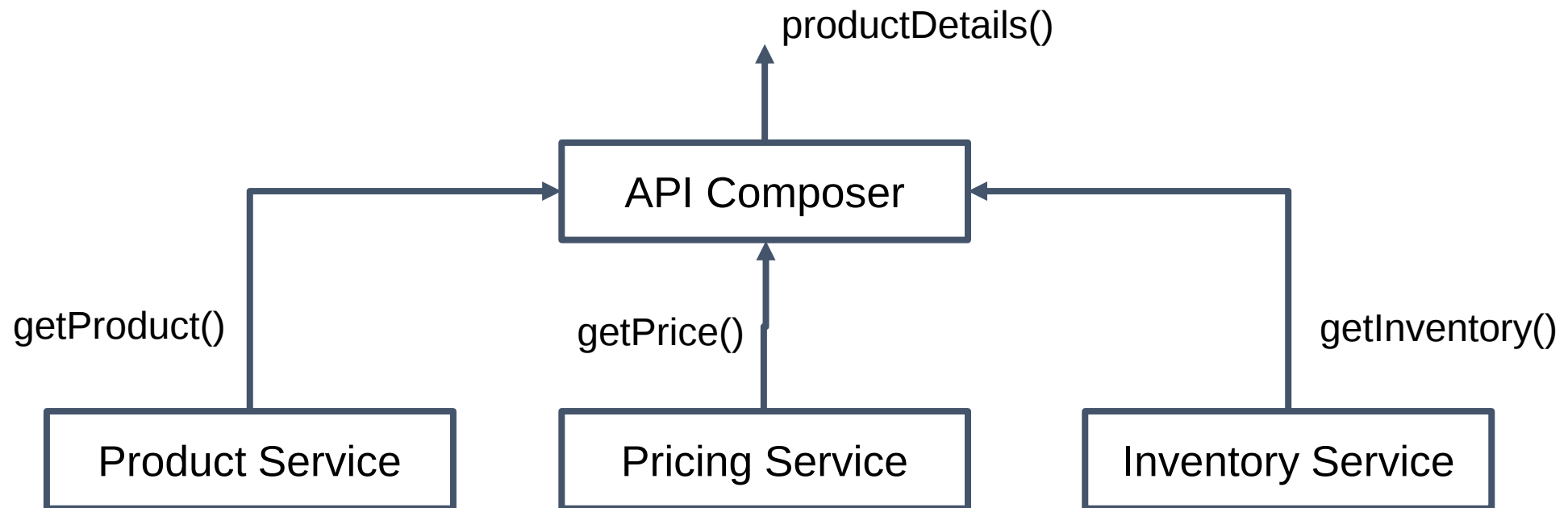
Two patterns to *retrieve* data scattered across multiple microservices:

(1) API Composition (e.g. through API Gateway)

(2) Replication

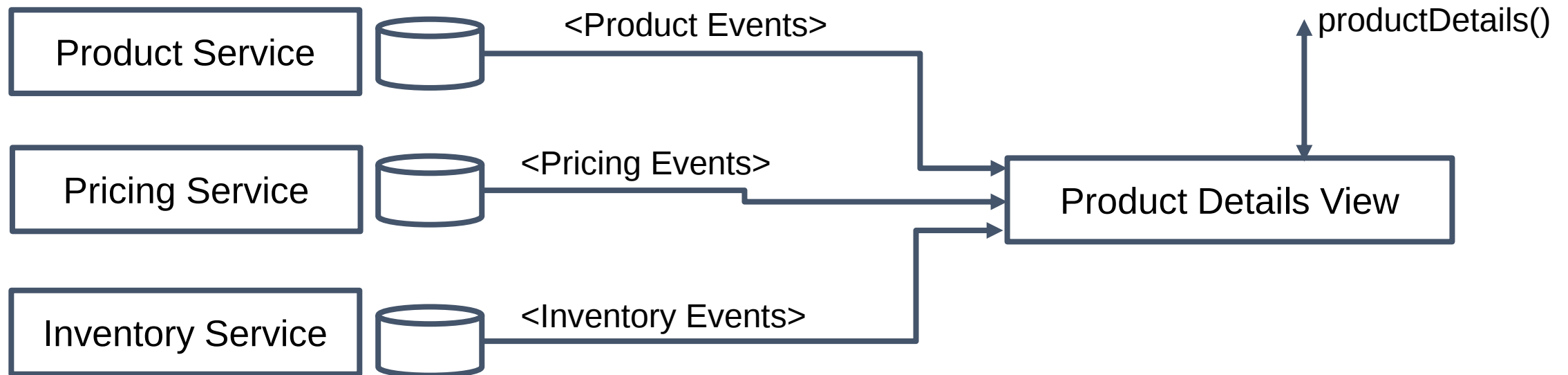
API Composition

The *API Composer* implements a query by invoking individual microservices that own the data, then combines the results with an in-memory join.



Replication

View / Read-only replica of data designed to support specific cross-service queries



Transactions Across Microservices

Some business transactions span multiple services

E.g.:

Customer places an order

Inventory is reserved

Payment Provider approves/denies payment

Order is confirmed / cancelled

Email Notification is sent

Order Service

Inventory
Service

Payment
Service

Email Service

Transactions Across Microservices

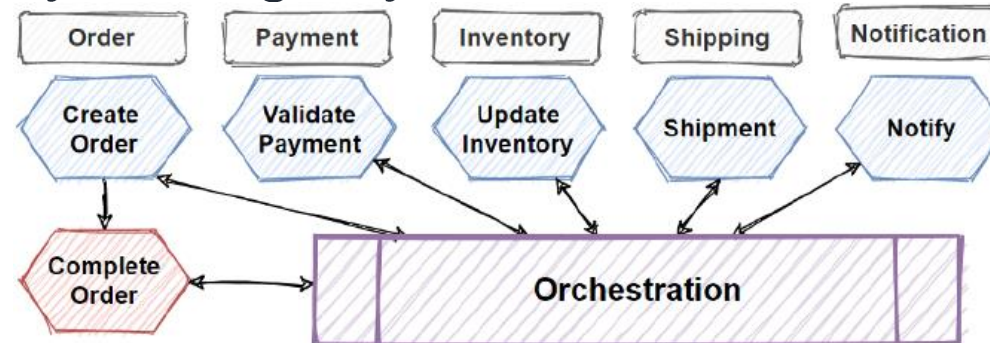
Two mechanism that enable distributed *transactions*

(1) Orchestration

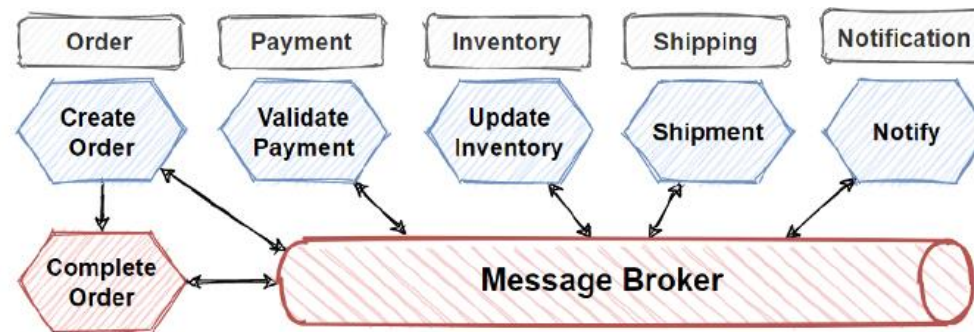
(2) Choreography

Mechanisms for Distributed Transactions in Microservices

Orchestration, typically through synchronous RPCs



Choreography, typically through asynchronous messages



<https://zongwb.medium.com/distributed-transactions-in-a-microservice-architecture-b4d6494de59e>

Agenda



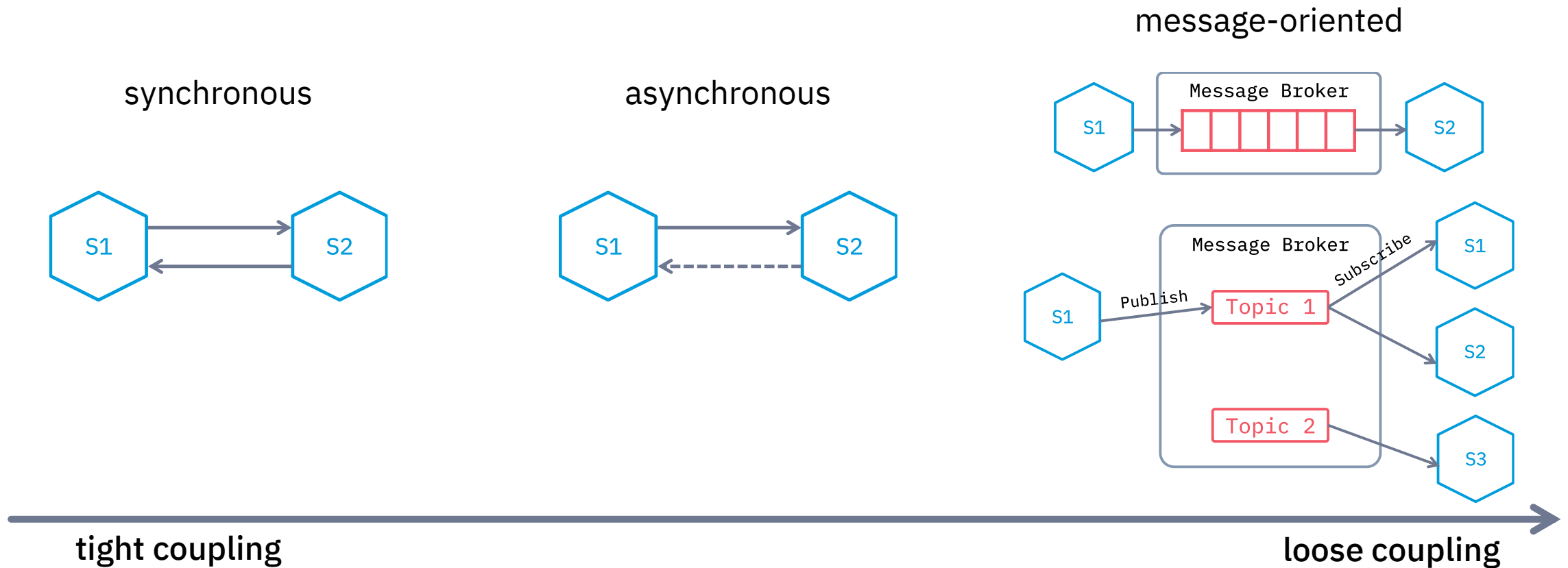
Microservices I: Characteristics and Design Principles

Microservices II: Using Microservices & Discussion

1. Data Management
2. Communication
3. API Design
4. Ifs and Buts of Microservices
5. Orga

Microservice Communication

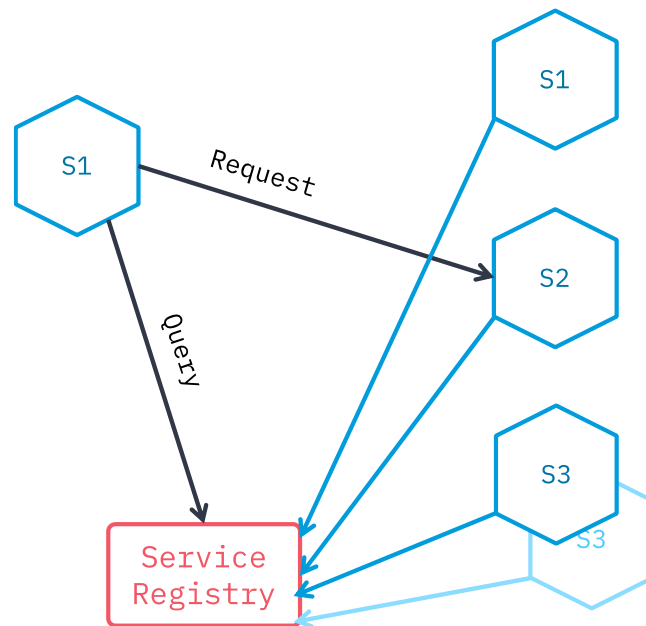
Loosening Availability Dependency



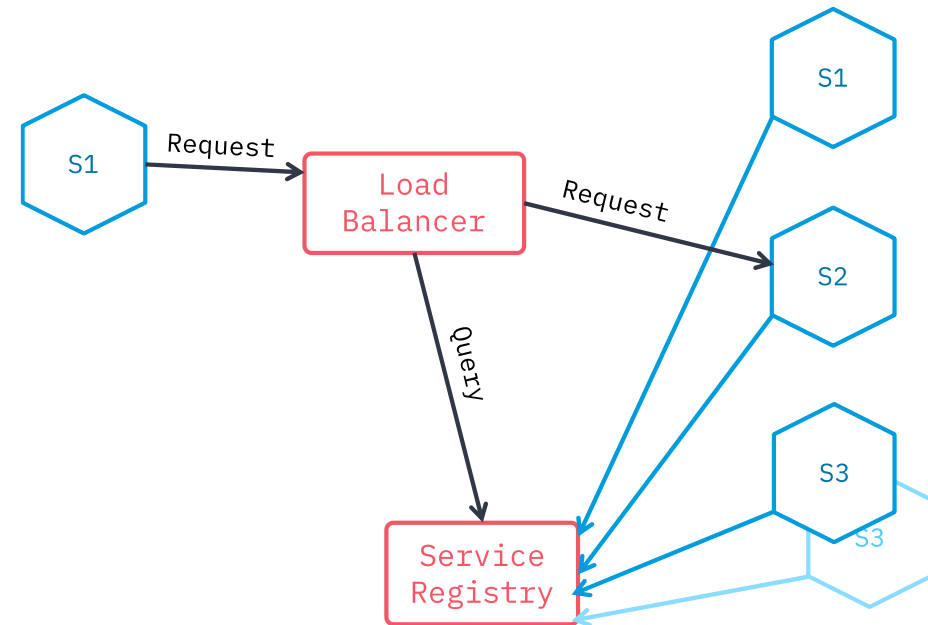
Microservice Communication

Loosening Location Dependency

client-side service discovery



server-side service discovery



tight coupling

loose coupling

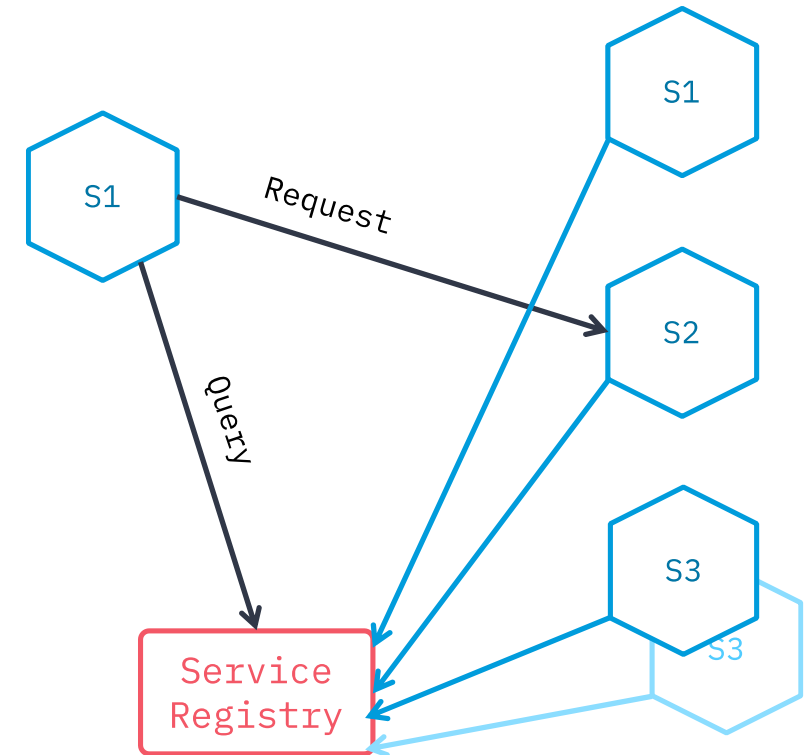
Client-side Service Discovery

The client is responsible for determining the network locations of available service instances and load balancing requests across them.

The client queries a service registry, which is a database of available service instances.

The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

E.g. Netflix's Eureka



<https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>

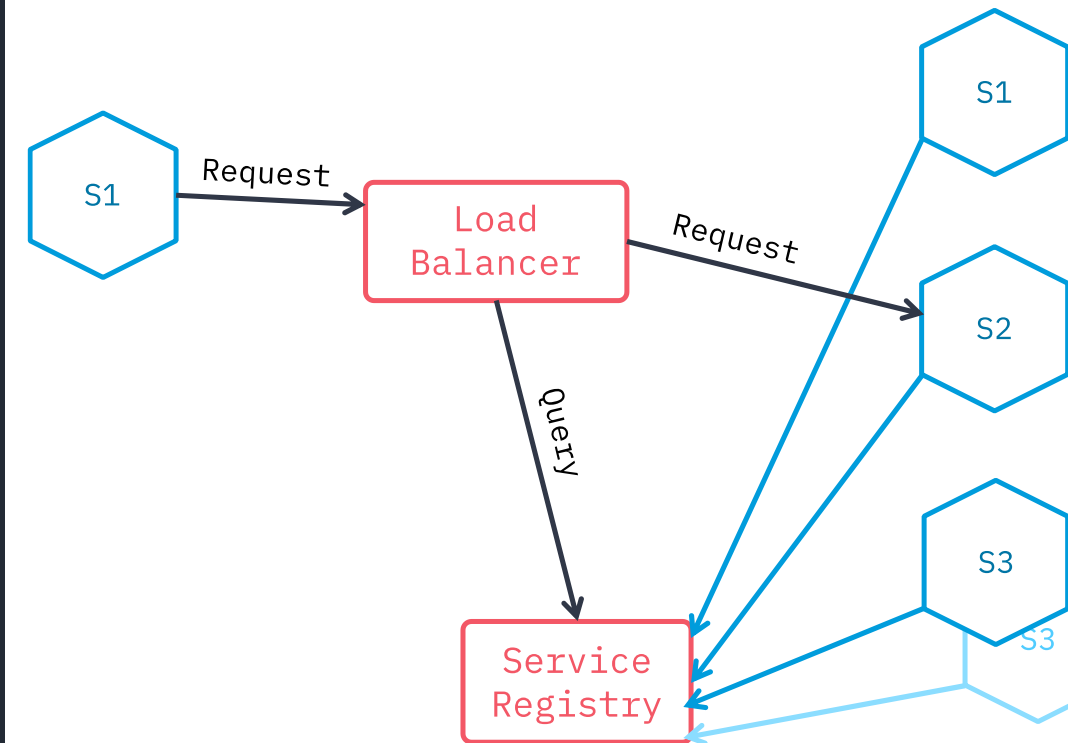
Server-side Service Discovery

The client makes a request to a service via a load balancer

The load balancer queries the service registry and routes each request to an available service instance

As with client-side discovery, service instances are registered and deregistered with the service registry

E.g. Kubernetes

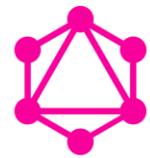


<https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>

Microservice Communication

Loosening Data Format Dependency?

{ REST }



GraphQL



gRPC

tight coupling

loose coupling

Recap: REST API

Representational State Transfer (REST) architectural style for distributed hypermedia systems (based on HTTP)

- Resource Identification
- Uniform Interface
- Self-Describing Messages
- HATEOAS
- Stateless Interactions
- Cache
- Layered System



Roy Fiedling (2000). Architectural Styles and the Design of Network-based Software Architectures.



<https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>

HTTP API Specification: OpenAPI

Open-source format for describing and documenting HTTP-based APIs

- Describes paths and resources of (ideally) RESTful APIs
- Used to generate code stubs for client and server for many languages
- Also serves as documentation

Both contract-first and code first-design are possible, the former is preferred

Reference: <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md>

REST Discussion

Strength

- Scalability
- Mashup-ability
- Usability (simplicity)
- Fixed set of methods
- Accessibility

Weaknesses

- Limited throughput
- Challenging to identify and locate resources appropriately in all applications
- Standards are not adopted

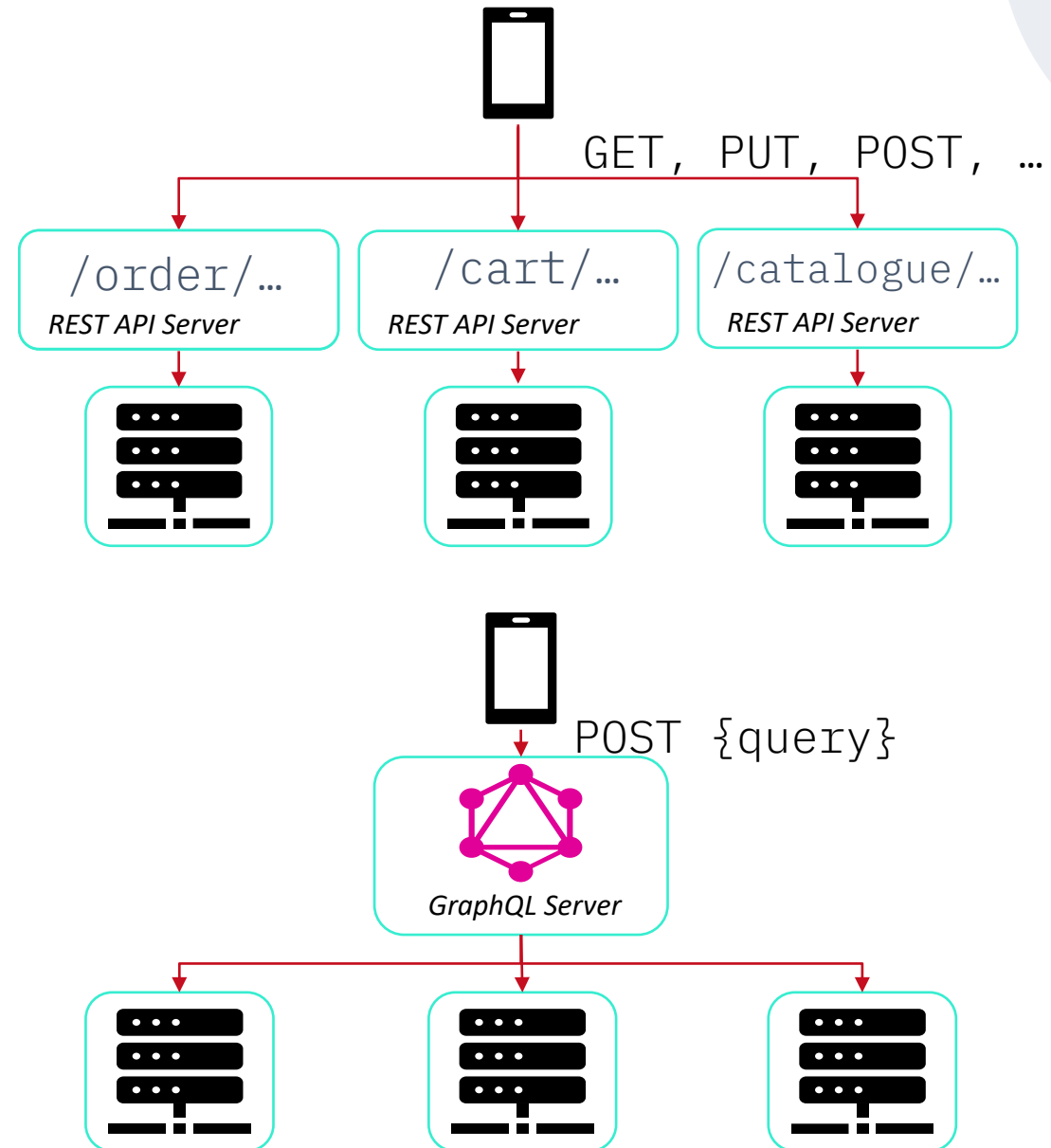
GraphQL

In **REST**, client request all the information in a resource, even if only parts are needed

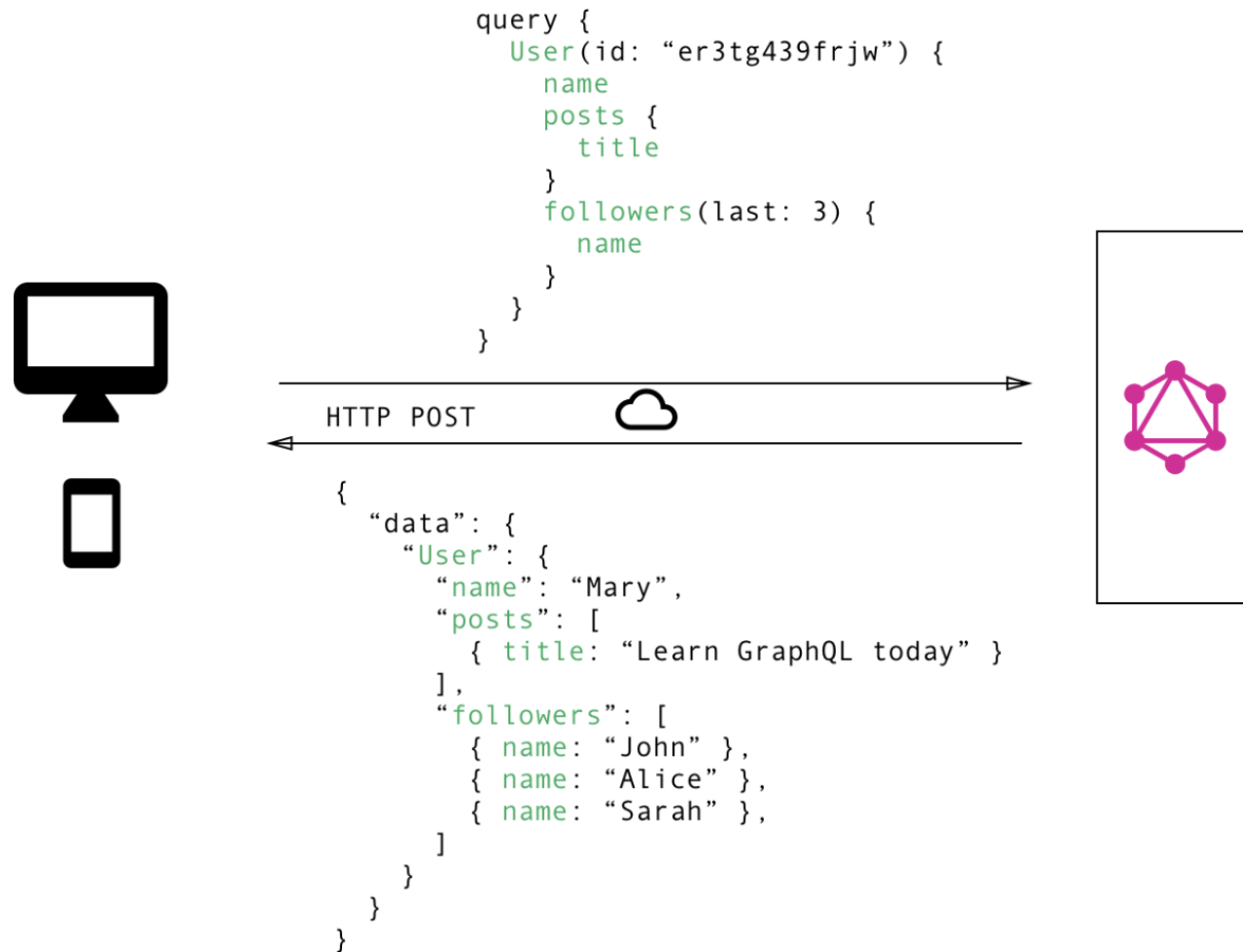
- “**Overfetching**” can slow down applications, especially if multiple endpoints are combined
- REST-clients need to know data structure and operations of all endpoint

In **GraphQL**, clients have the ability to dictate exactly what they need from the server

- **Separation of concerns**: client needs to know about data requirements, server resolves it
- Point of **integration is shifted to server**



GraphQL - Example



<https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>

GraphQL Discussion

Strength

- Declarative data fetching
- No overfetching
- Inspect types and fields at runtime
- Gateway for different APIs
- Updates are easy (new fields possible, @deprecated)

Weaknesses

- Caching, rate limiting, error handling and other features common in the Web are not trivial here
- Nested queries can easily lead to performance issues
- Understanding of large schema
- Private fields are not supported

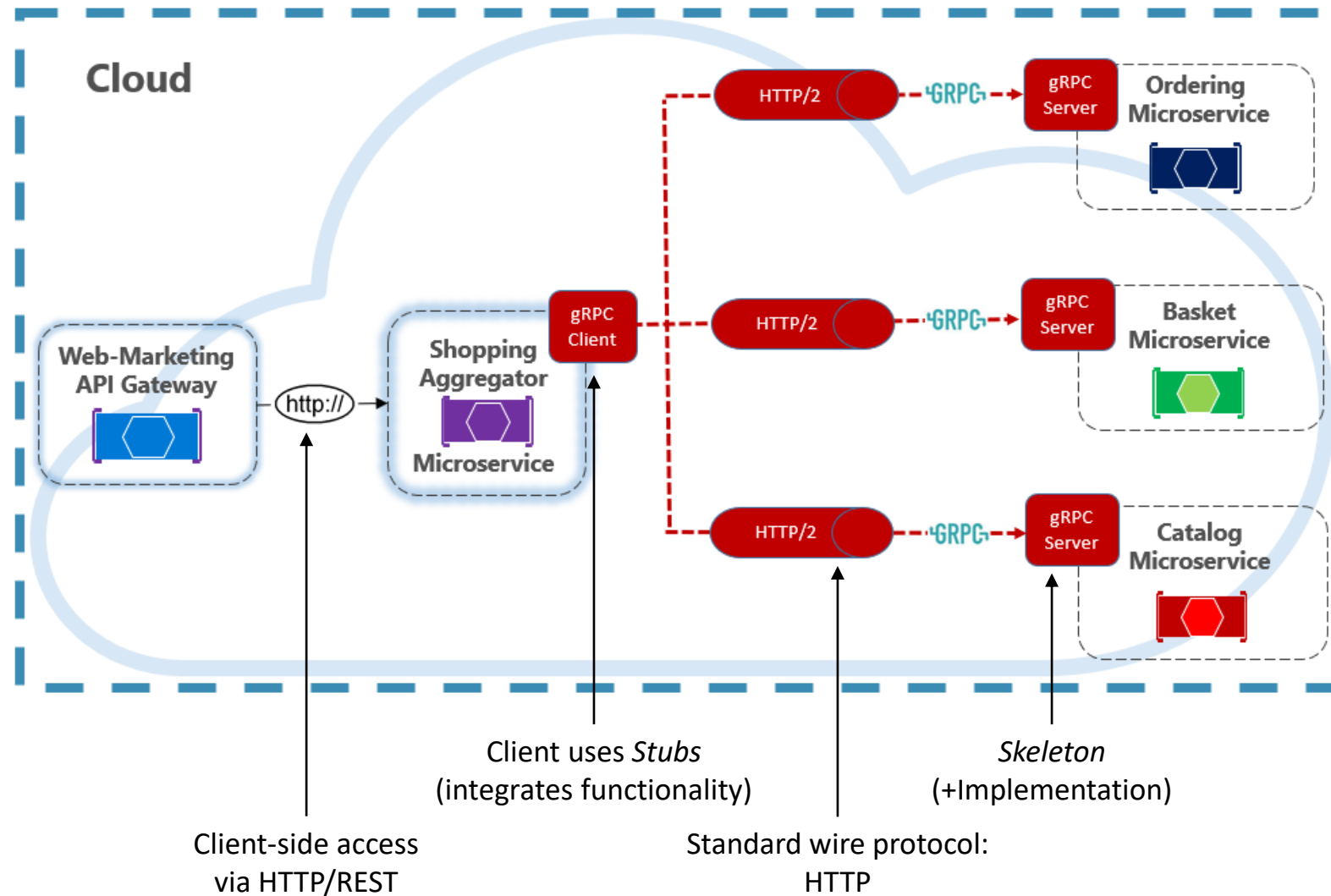
RPC Frameworks

- They focus on efficient serialization and often are agnostic regarding transport protocol
 - Transport protocol can be standard HTTP or a custom implementation
 - Serialization is often into efficient, compact binary representation
- Integrated functionality is kept minimal
 - Encryption and authentication are common
 - Everything else: plugins



Thrift

gRPC - Example



<https://docs.microsoft.com/de-de/dotnet/architecture/cloud-native/media/grpc-implementation.png>

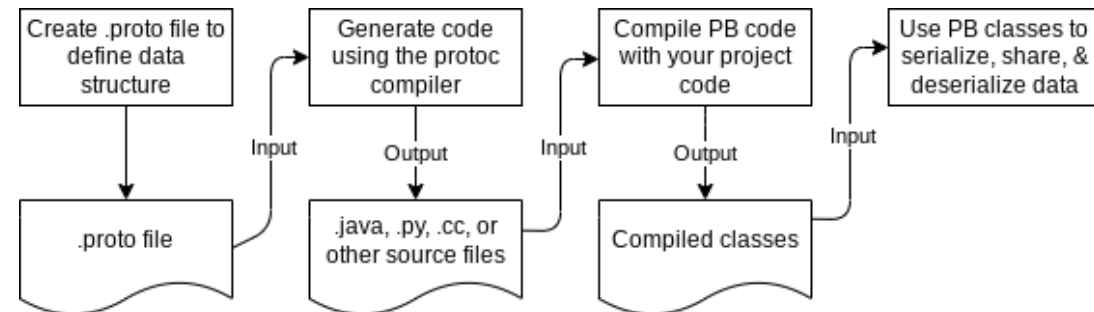
Protocol Buffers

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for **serializing structured data** in a **forward-compatible** and **backward-compatible** way.

Introduced by Google in 2008 → today Proto3

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

A proto definition (.proto)



Protocol buffers workflow.

<https://protobuf.dev/overview/#updating-defs>

gRPC Discussion

Strength

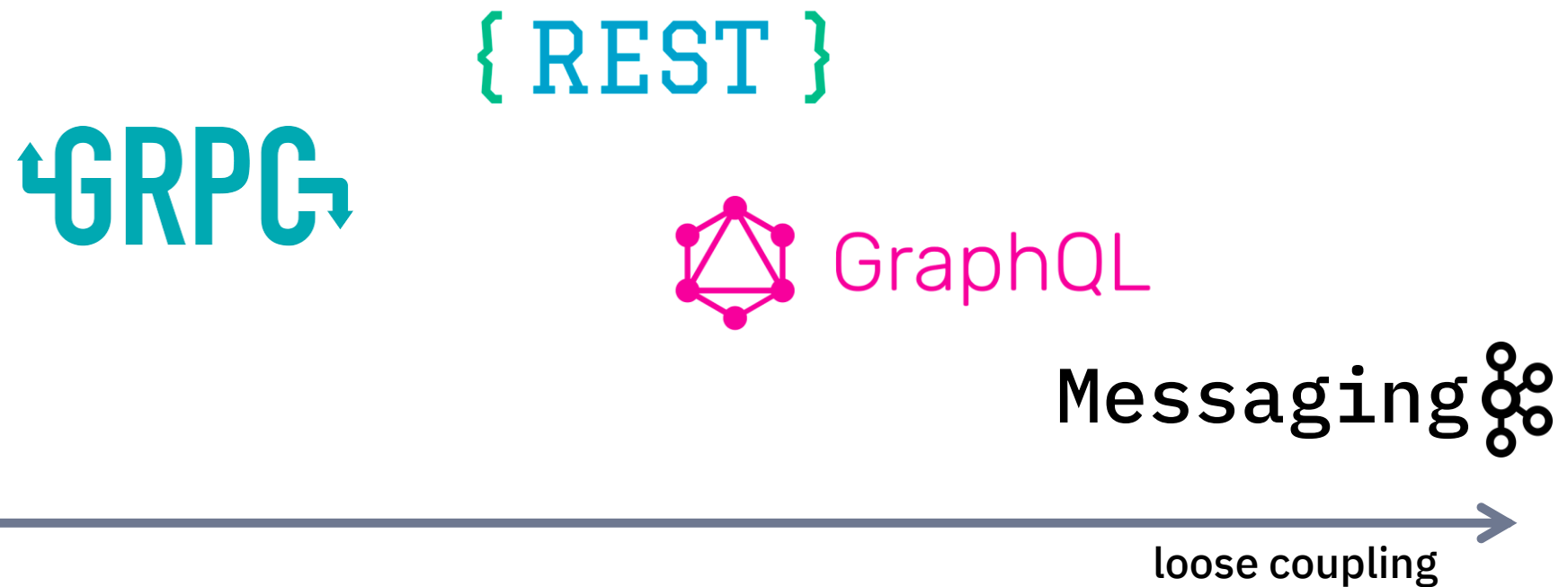
- High throughput
- Handle requests simultaneously
- Low processing complexity
- Typed contracts
- Serialized messages

Weaknesses

- Small message payload
- Overwhelming API: developers have to learn every method
→ time-consuming & error-prone
- Limited browser support (gRPC-web)

Microservice Communication

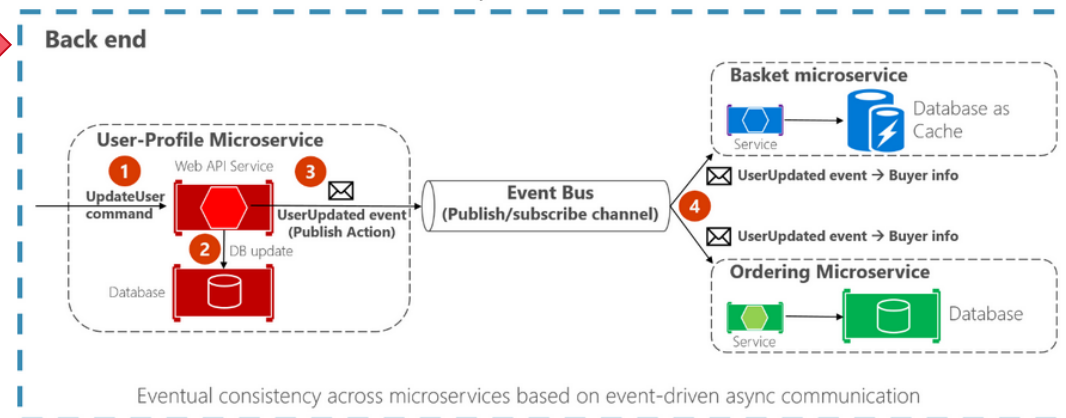
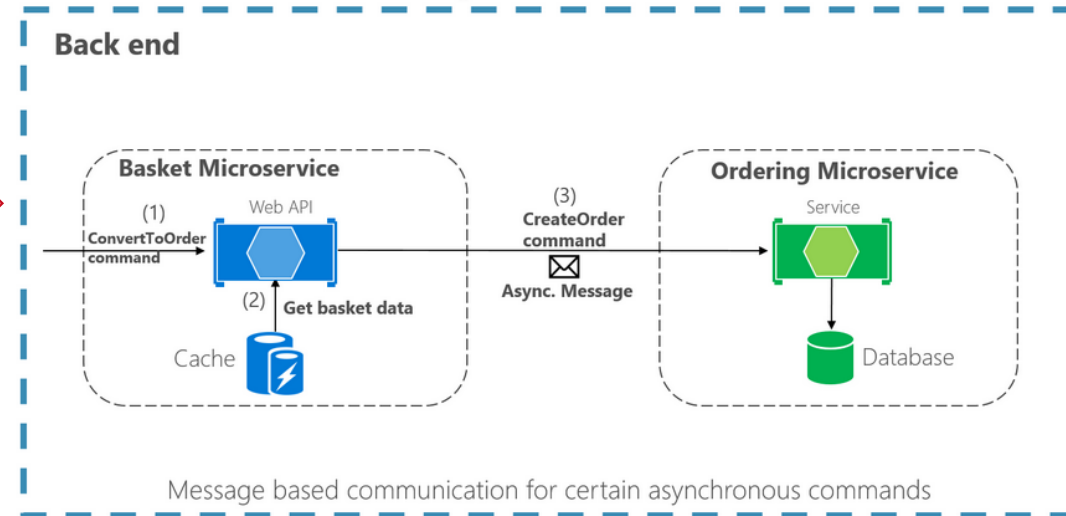
Loosening Data Format Dependency?



Messaging

Single-receiver: a message is sent over a channel and processed exactly once

Multiple-receiver: to gain more flexibility, messages are triggered by an event, sent over a channel (e.g. an event bus) and consumed by subscribers (pub/sub)

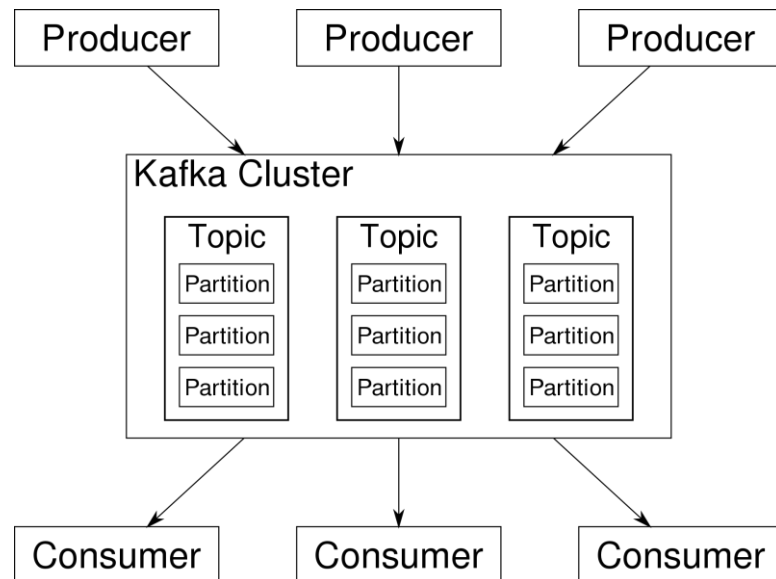


<https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication>

Messaging Example: Apache Kafka

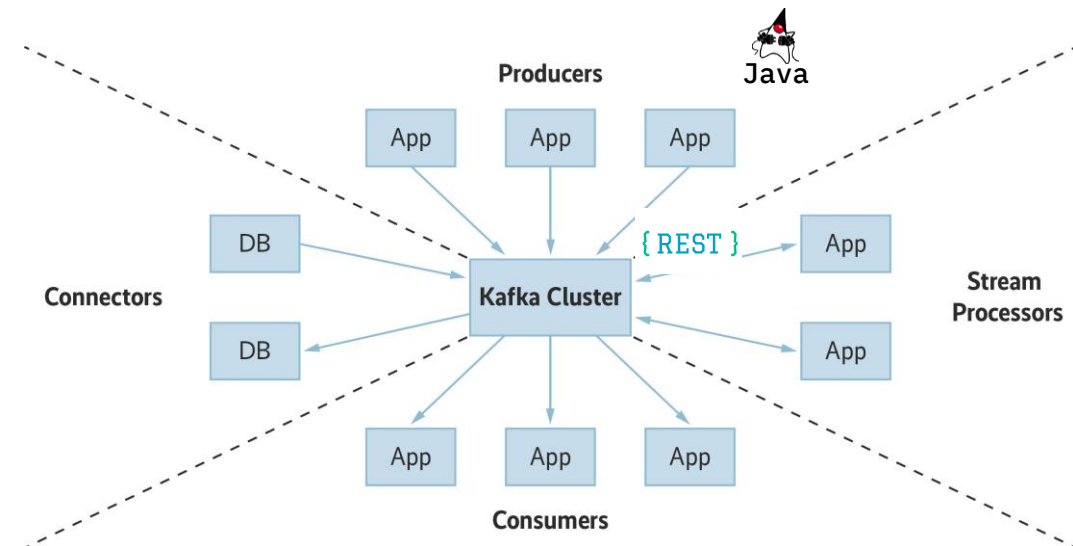
Messaging middleware, like Kafka, can rely on custom or standard protocols to interact with the cluster

Apache Kafka Architecture



<https://commons.wikimedia.org/w/index.php?curid=59871096>

- Kafka is a „distributed event streaming platform“
- Producers push messages, consumers poll for messages



<https://www.heise.de/select/ix/2019/4/1553935521978043>

Apache Kafka Discussion

Strength

- High throughput
 - Low latency
 - Scalability
 - Near real time
 - Adding/ removing services is easy
 - Individual pull-rates
- Well suited for internal services

Weaknesses

- Need more partitions than consumers (idling)
- Managing/monitoring not included
- ZooKeeper as bottleneck
- Not suitable for front-end services

Inter-Microservice Coordination in Practice

- Survey Results

- Often custom-built solutions
- *“Persistent, asynchronous message-passing is the most used type of communication (42% of respondents) in cases where an operation spans multiple microservices.”*
- *“Synchronous communication (39.5% of respondents), mostly through REST APIs, is often used to serve client requests, such as queries or updates to data items.”*

Source: doi.org/10.14778/3484224.3484232

Summary: Microservice Communication

There are several communication methods that can be used in a microservice architecture:

- HTTP-based (e.g. REST)
- RPC-based (e.g. gRPC)
- Messaging-based (e.g. Apache Kafka)

All of them can be beneficial, it is highly dependend on your application needs which one you should choose.

Agenda



Microservices I: Characteristics and Design Principles

Microservices II: Using Microservices & Discussion

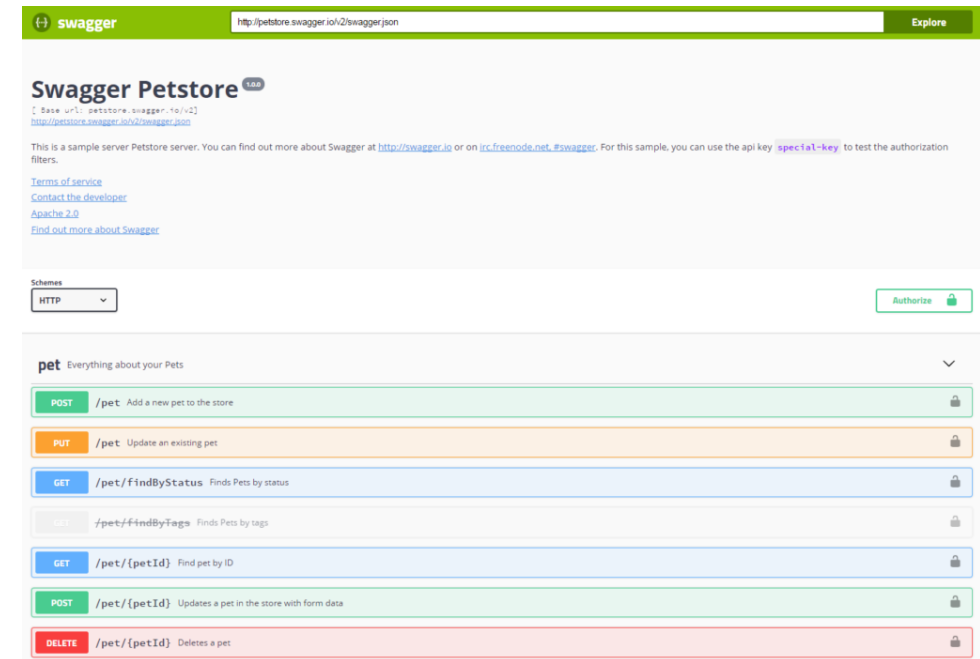
1. Data Management
2. Communication
3. API Design
4. Ifs and Buts of Microservices
5. Orga

*APIs can be among your greatest assets
or liabilities.*

Joshua Bloch (2006)

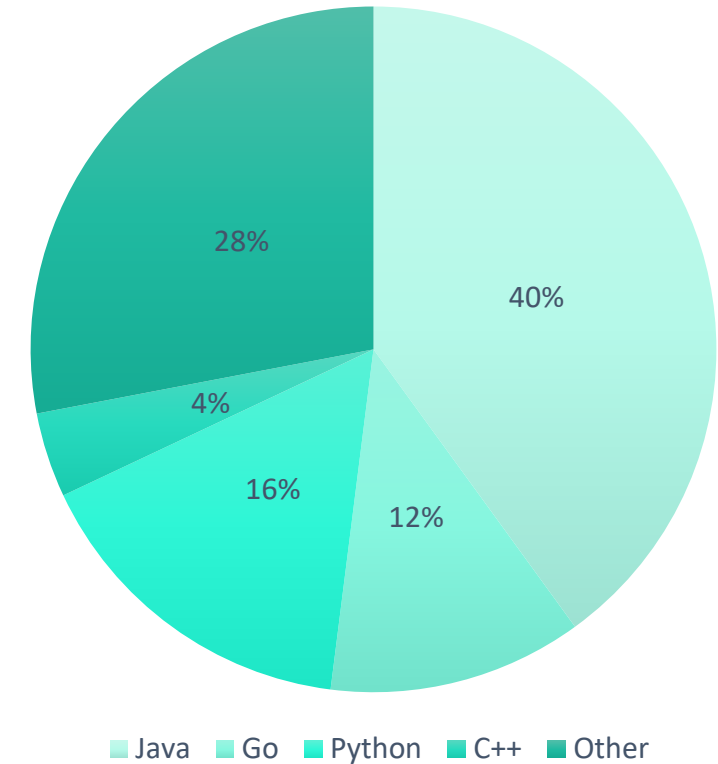
API-driven development

- “API first” – specify an interface, then implement
- Internal vs. External APIs
- API-level documentation
- API documentation serves as contract
 - between teams
 - for testing
- API versioning makes changes explicit



Technologies used for API Implementation

- Java-based Technologies
 - Spring *
- Go-based Technologies
 - Go-Kit
 - Gizmo
- Python-based Technologies
 - Python Flask
 - Django
 - Nameko
- C++
 - C++ REST SDK
- Other
 - JWT
 - Swagger



Fangwei Chen, Li Zhang, and Xiaoli Lian (2021). A systematic gray literature review: The technologies and concerns of microservice application programming interfaces. In: *Software: Practice and Experience*

API Versioning

Versioning is challenging for RESTful APIs. The following versioning strategies can be used:

- URI: /api/v1/example
 - Pro: easy for clients
 - Con: changes to code-base imply new API branch
- Query parameter
 - Pro: easy to implement
 - Con: routing requests gets more complicated
- Custom header (accept-header/request header)
 - Pro: no need to change URI
 - Con: define custom header
- Content Negotiation
 - Pro: resource version instead of API version
 - Con: less accessible

API Design Maxims I

- APIs should be easy to use and hard to misuse.
- APIs should be self-documenting.
- Public APIs, like diamonds, are forever → When in doubt, leave it out.
- Structure requirements as use-cases; code the use-cases against your API before you implement it; maintain the code for use-cases.
- API design is not a solitary activity.
- Fail fast.



Bloch, Joshua. "How to design a good API and why it matters." *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006.

API Design Maxims II

- Example code should be exemplary.
- Avoid fixed limits on input sizes.
- Overload with care.
- Use consistent parameter ordering across methods.
- Avoid long parameter lists.
- Throw exceptions only to indicate exceptional conditions.



Bloch, Joshua. "How to design a good API and why it matters." *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006.

Microsoft Best Practice

A well-designed web API should support platform independence and service evolution.

APIS should be designed around resources: resource URIs should be based on nouns (/orders), not verbs (/create-order).

Avoid URIs that are more complex than *collection/item/collection*.

Avoid „chatty web APIs“: denormalize the data and minimize the number of requests (decrease load on server) – however, keep overfetching in mind (latency/bandwidth costs).

Avoid dependencies between API and underlying data sources.

<https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>

Google Best Practices

Naming should be **simple, intuitive, and consistent**:

- Names used in APIs should be in correct American English: *color* vs *colour*.
- Commonly accepted short forms or abbreviations of long words may be used for brevity: *API* vs *Application Programming Interface*.
- Use intuitive, familiar terminology where possible: *delete* vs *erase*.
- Use the same name or term for the same concept, including for concepts shared across APIs.
- Avoid name overloading. Use different names for different concepts.

https://cloud.google.com/apis/design/naming_convention

Agenda



Microservices I: Characteristics and Design Principles

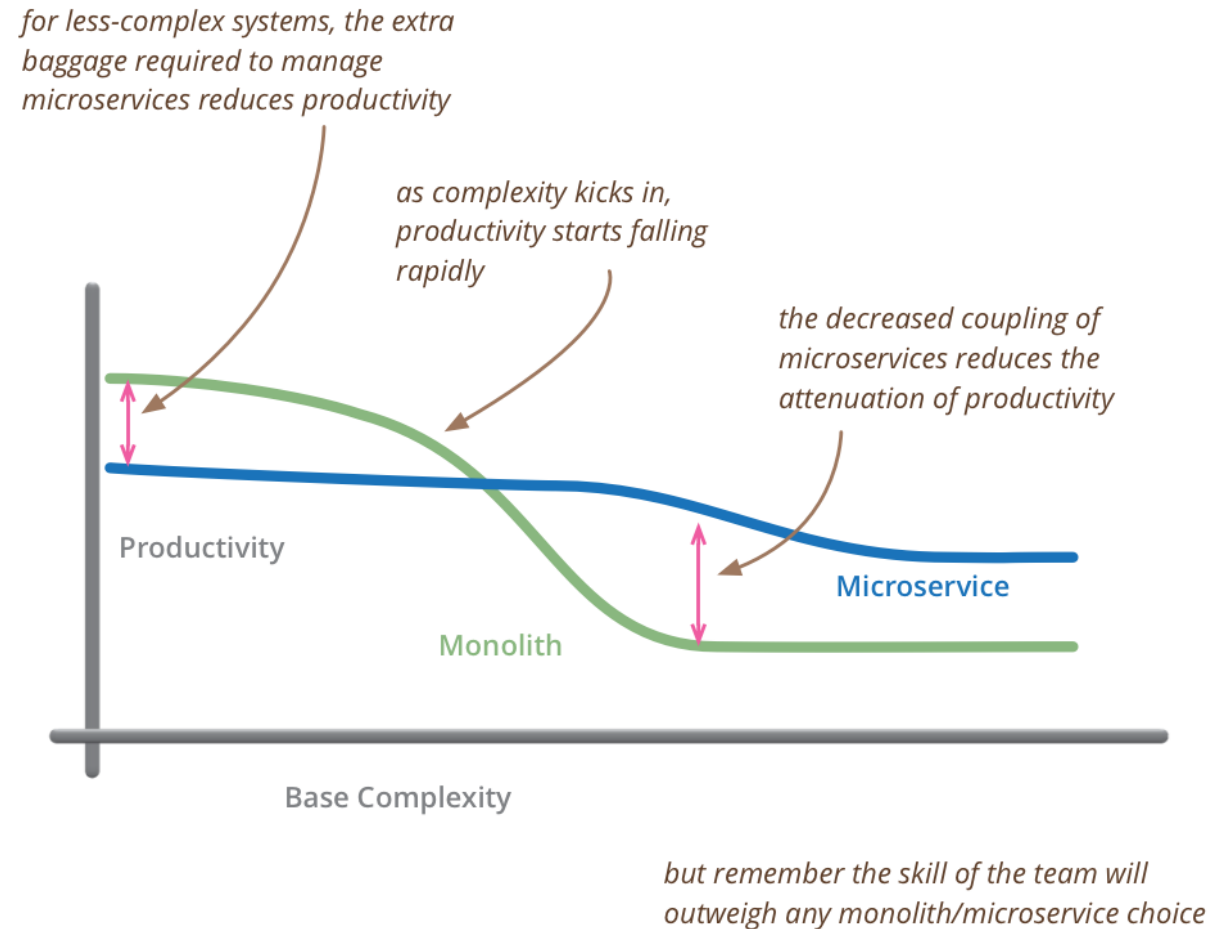
Microservices II: Using Microservices & Discussion

1. Data Management
2. Communication
3. API Design
4. Ifs and Buts of Microservices
5. Orga

MicroservicesPremium

The increased complexity introduced through microservices is referred to as microservice premium.

The majority of software systems should be built as a single (modular) monolithic application.



Source: <https://martinfowler.com/bliki/MicroservicePremium.html>

Common Pitfalls

- Carelessness: Scalability and stability are assumed to be provided by default, however, this is only true if all characteristics are addressed.
- Service orphans: Small services still require documentation, testing and verification, and maintenance.
- Failure tolerance is not natively given: each service needs to be able to recover from failures.
- One hammer: evaluate if microservices are adequate for your application and organisation.
- Forget about qualities: adding *ilities late in the process is expensive.

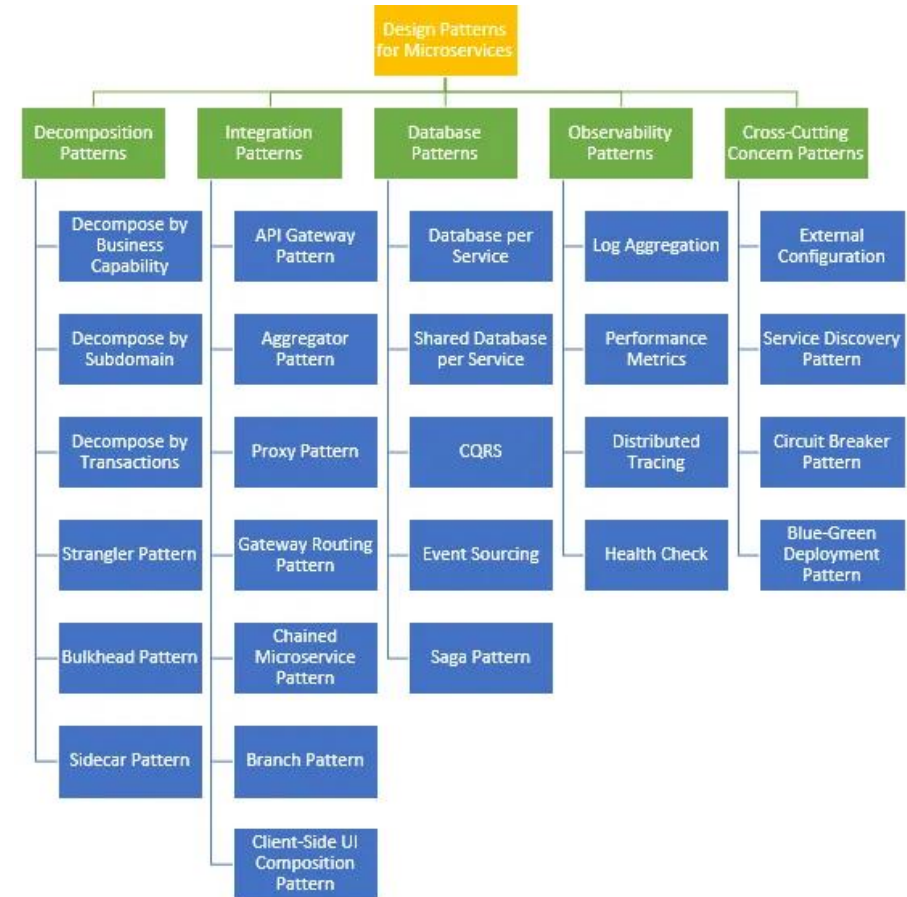


Marcus Hilbrich and Fabian Lehmann. "Discussing Microservices: Definitions, Pitfalls, and their Relations." *2022 IEEE International Conference on Services Computing (SCC)*. 2022.

Addressing Qualities

Availability
Consistency
Decentralized governance
Decoupled
Flexibility
Independent, autonomous
Privacy
Resilience
Scalability
Security
...

e.g. through design patterns



<https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a>

CAP Theorem



Microservice architectures always ensure **partition tolerance**.
Which other quality/ies can be ensured (using the CAP theorem)?

Microservices can either address **consistency** or **availability**,
but not both at the same time.

→ Qualities can be conflicting!

Agenda



Microservices I: Characteristics and Design Principles

Microservices II: Using Microservices & Discussion

1. Data Management
2. Communication
3. API Design
4. Ifs and Buts of Microservices
5. Orga

Formal registration deadline: 15.05.2023, 23:59

The registration via moses: <https://moseskonto.tu-berlin.de/moses/modultransfersystem/index.html>

- select "MTS / Module exams registration/deregistration" after logging in
- Select right course and semester!
- instructions with screenshots are only available in [German](#)

How to handle service versioning and backward compatibility in a
Microservices architecture?

CNAE Schedule (preliminary)

